TCP/IP Illustrated Volu

The P

W. tevens

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

TCP/IP Illustrated, Volume 1

The Protocols

W. Richard Stevens

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Stevens, W. Richard
TCP/IP Illustrated: the protocols/W. Richard Stevens.
p. cm.–(Addison-Wesley professional computing series)
Includes bibliographical references and index.
ISBN 0-201-63346-9 (v.1)
1.TCP/IP (Computer network protocol) I. Title. II. Series.
TK5105.55S74 1994
004.6'2—dc20

Text printed on recycled and acid-free paper.

ISBN 0201633469
19 2021222324 MA 04 03 02 01
19th Printing July 2001

3.2 IP Header

Figure 3.1 shows the format of an IP datagram. The normal size of the IP header is 20 bytes, unless options are present.

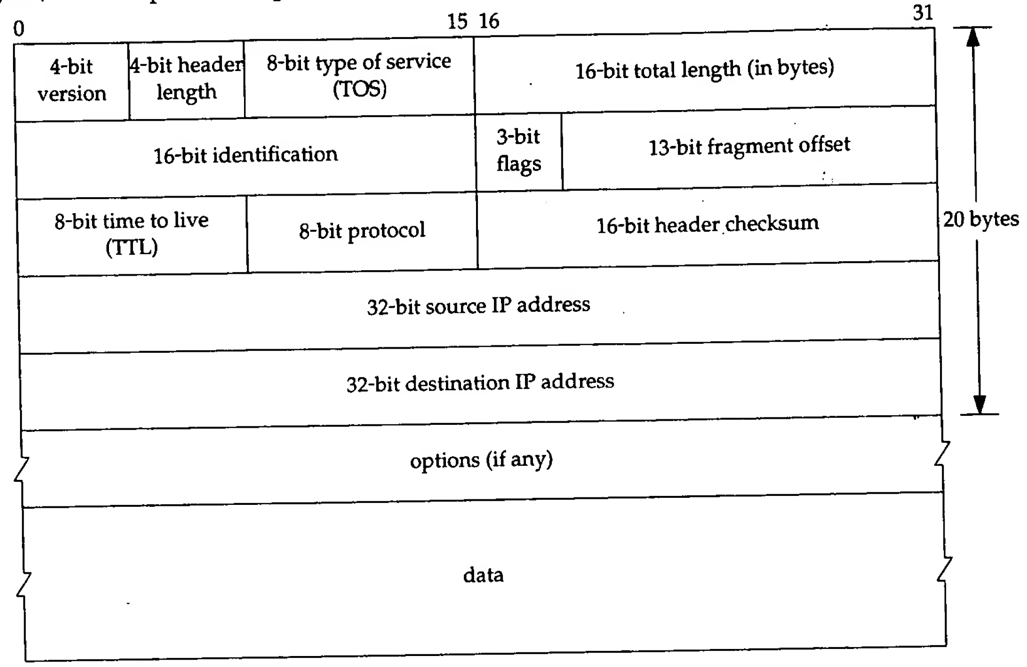

Figure 3.1 IP datagram, showing the fields in the IP header.

We will show the pictures of protocol headers in TCP/IP as in Figure 3.1. The most significant bit is numbered 0 at the left, and the least significant bit of a 32-bit value is numbered 31 on the right.

The 4 bytes in the 32-bit value are transmitted in the order: bits 0–7 first, then bits 8–15, then 16–23, and bits 24–31 last. This is called *big endian* byte ordering, which is the byte ordering required for all binary integers in the TCP/IP headers as they traverse a network. This is called the *network byte order*. Machines that store binary integers in other formats, such as the *little endian* format, must convert the header values into the network byte order before transmitting the data.

The current protocol *version* is 4, so IP is sometimes called IPv4. Section 3.10 discusses some proposals for a new version of IP.

The *header length* is the number of 32-bit words in the header, including any options. Since this is a 4-bit field, it limits the header to 60 bytes. In Chapter 8 we'll see that this limitation makes some of the options, such as the record route option, useless today. The normal value of this field (when no options are present) is 5.

The *type-of-service* field (TOS) is composed of a 3-bit precedence field (which is ignored today), 4 TOS bits, and an unused bit that must be 0. The 4 TOS bits are: minimize delay, maximize throughput, maximize reliability, and minimize monetary cost.

Only 1 of these 4 bits can be turned on. If all 4 bits are 0 it implies normal service. RFC 1340 [Reynolds and Postel 1992] specifies how these bits should be set by all the standard applications. RFC 1349 [Almquist 1992] contains some corrections to this RFC, and a more detailed description of the TOS feature.

Figure 3.2 shows the recommended values of the TOS field for various applications. In the final column we show the hexadecimal value, since that's what we'll see in the `tcpdump` output later in the text.

Application	Minimize delay	Maximize throughput	Maximize reliability	Minimize monetary cost	Hex value
Telnet/Rlogin	1	0	0	0	0x10
FTP					
control	1	0	0	0	0x10
data	0	1	0	0	0x08
any bulk data	0	1	0	0	0x08
TFTP	1	0	0	0	0x10
SMTP					
command phase	1	0	0	0	0x10
data phase	0	1	0	0	0x08
DNS					
UDP query	1	0	0	0	0x10
TCP query	0	0	0	0	0x00
zone transfer	0	1	0	0	0x08
ICMP					
error	0	0	0	0	0x00
query	0	0	0	0	0x00
any IGP	0	0	1	0	0x04
SNMP	0	0	1	0	0x04
BOOTP	0	0	0	0	0x00
NNTP	0	0	0	1	0x02

Figure 3.2 Recommended values for type-of-service field.

The interactive login applications, Telnet and Rlogin, want a minimum delay since they're used interactively by a human for small amounts of data transfer. File transfer by FTP, on the other hand, wants maximum throughput. Maximum reliability is specified for network management (SNMP) and the routing protocols. Usenet news (NNTP) is the only one shown that wants to minimize monetary cost.

The TOS feature is not supported by most TCP/IP implementations today, though newer systems starting with 4.3BSD Reno are setting it. Additionally, new routing protocols such as OSPF and IS–IS are capable of making routing decisions based on this field.

In Section 2.10 we mentioned that SLIP drivers normally provide type-of-service queueing, allowing interactive traffic to be handled before bulk data. Since most implementations don't use the TOS field, this queueing is done ad hoc by SLIP, with the driver looking at the protocol field (to determine whether it's a TCP segment or not) and then checking the source and destination TCP port numbers to see if the port number corresponds to an interactive service. One driver comments that this "disgusting hack" is required since most implementations don't allow the application to set the TOS field.

The *total length* field is the total length of the IP datagram in bytes. Using this field and the header length field, we know where the data portion of the IP datagram starts, and its length. Since this is a 16-bit field, the maximum size of an IP datagram is 65535 bytes. (Recall from Figure 2.5 [p. 30] that a Hyperchannel has an MTU of 65535. This means there really isn't an MTU—it uses the largest IP datagram possible.) This field also changes when a datagram is fragmented, which we describe in Section 11.5.

Although it's possible to send a 65535-byte IP datagram, most link layers will fragment this. Furthermore, a host is not required to receive a datagram larger than 576 bytes. TCP divides the user's data into pieces, so this limit normally doesn't affect TCP. With UDP we'll encounter numerous applications in later chapters (RIP, TFTP, BOOTP, the DNS, and SNMP) that limit themselves to 512 bytes of user data, to stay below this 576-byte limit. Realistically, however, most implementations today (especially those that support the Network File System, NFS) allow for just over 8192-byte IP datagrams.

The total length field is required in the IP header since some data links (e.g., Ethernet) pad small frames to be a minimum length. Even though the minimum Ethernet frame size is 46 bytes (Figure 2.1), an IP datagram can be smaller. If the total length field wasn't provided, the IP layer wouldn't know how much of a 46-byte Ethernet frame was really an IP datagram.

The *identification* field uniquely identifies each datagram sent by a host. It normally increments by one each time a datagram is sent. We return to this field when we look at fragmentation and reassembly in Section 11.5. Similarly, we'll also look at the *flags* field and the *fragmentation offset* field when we talk about fragmentation.

> RFC 791 [Postel 1981a] says that the identification field should be chosen by the upper layer that is having IP send the datagram. This implies that two consecutive IP datagrams, one generated by TCP and one generated by UDP, can have the same identification field. While this is OK (the reassembly algorithm handles this), most Berkeley-derived implementations have the IP layer increment a kernel variable each time an IP datagram is sent, regardless of which layer passed the data to IP to send. This kernel variable is initialized to a value based on the time-of-day when the system is bootstrapped.

The *time-to-live* field, or *TTL*, sets an upper limit on the number of routers through which a datagram can pass. It limits the lifetime of the datagram. It is initialized by the sender to some value (often 32 or 64) and decremented by one by every router that handles the datagram. When this field reaches 0, the datagram is thrown away, and the sender is notified with an ICMP message. This prevents packets from getting caught in routing loops forever. We return to this field in Chapter 8 when we look at the Traceroute program.

We talked about the *protocol* field in Chapter 1 and showed how it is used by IP to demultiplex incoming datagrams in Figure 1.8. It identifies which protocol gave the data for IP to send.

The *header checksum* is calculated over the IP header only. It does *not* cover any data that follows the header. ICMP, IGMP, UDP, and TCP all have a checksum in their own headers to cover their header and data.

To compute the IP checksum for an outgoing datagram, the value of the checksum field is first set to 0. Then the 16-bit one's complement sum of the header is calculated (i.e., the entire header is considered a sequence of 16-bit words). The 16-bit one's complement of this sum is stored in the checksum field. When an IP datagram is received, the 16-bit one's complement sum of the header is calculated. Since the receiver's calculated checksum contains the checksum stored by the sender, the receiver's checksum is all one bits if nothing in the header was modified. If the result is not all one bits (a checksum error), IP discards the received datagram. No error message is generated. It is up to the higher layers to somehow detect the missing datagram and retransmit.

ICMP, IGMP, UDP, and TCP all use the same checksum algorithm, although TCP and UDP include various fields from the IP header, in addition to their own header and data. RFC 1071 [Braden, Borman, and Partridge 1988] contains implementation techniques for computing the Internet checksum. Since a router often changes only the TTL field (decrementing it by 1), a router can incrementally update the checksum when it forwards a received datagram, instead of calculating the checksum over the entire IP header again. RFC 1141 [Mallory and Kullberg 1990] describes an efficient way to do this.

The standard BSD implementation, however, does not use this incremental update feature when forwarding a datagram.

Every IP datagram contains the *source IP address* and the *destination IP address*. These are the 32-bit values that we described in Section 1.4.

The final field, the *options*, is a variable-length list of optional information for the datagram. The options currently defined are:

- security and handling restrictions (for military applications, refer to RFC 1108 [Kent 1991] for details),
- record route (have each router record its IP address, Section 7.3),
- timestamp (have each router record its IP address and time, Section 7.4),
- loose source routing (specifying a list of IP addresses that must be traversed by the datagram, Section 8.5), and
- strict source routing (similar to loose source routing but here only the addresses in the list can be traversed, Section 8.5).

These options are rarely used and not all host and routers support all the options.

The options field always ends on a 32-bit boundary. Pad bytes with a value of 0 are added if necessary. This assures that the IP header is always a multiple of 32 bits (as required for the *header length* field).

3.3 IP Routing

Conceptually, IP routing is simple, especially for a host. If the destination is directly connected to the host (e.g., a point-to-point link) or on a shared network (e.g., Ethernet or token ring), then the IP datagram is sent directly to the destination. Otherwise the source mode (`-i`) to send data instead of trying to read and write standard input and output. The `-n4` option says to output 4 datagrams (instead of the default 1024) and the destination host is `svr4`. We described the discard service in Section 1.12. We use the default output size of 1024 bytes per write.

The second time we run the program we specify `-w0`, causing 0-length datagrams to be written. Figure 11.6 shows the `tcpdump` output for both commands.

```
1   0.0                    bsdi.1108 > svr4.discard: udp 1024
2   0.002424  ( 0.0024)    bsdi.1108 > svr4.discard: udp 1024
3   0.006210  ( 0.0038)    bsdi.1108 > svr4.discard: udp 1024
4   0.010276  ( 0.0041)    bsdi.1108 > svr4.discard: udp 1024

5  41.720114  (41.7098)    bsdi.1110 > svr4.discard: udp 0
6  41.721072  ( 0.0010)    bsdi.1110 > svr4.discard: udp 0
7  41.722094  ( 0.0010)    bsdi.1110 > svr4.discard: udp 0
8  41.723070  ( 0.0010)    bsdi.1110 > svr4.discard: udp 0
```

Figure 11.6 `tcpdump` output when UDP datagrams are sent in one direction.

This output shows the four 1024-byte datagrams, followed by the four 0-length datagrams. Each datagram followed the previous by a few milliseconds. (It took 41 seconds to type in the second command.)

There is no communication between the sender and receiver before the first datagram is sent. (We'll see in Chapter 17 that TCP must establish a connection with the other end before the first byte of data can be sent.) Also, there are no acknowledgments by the receiver when the data is received. The sender, in this example, has no idea whether the other end receives the datagrams.

Finally note that the source UDP port number changes each time the program is run. First it is 1108 and then it is 1110. We mentioned in Section 1.9 that the ephemeral port numbers used by clients are typically in the range 1024 through 5000, as we see here.

11.5 IP Fragmentation

As we described in Section 2.8, the physical network layer normally imposes an upper limit on the size of the frame that can be transmitted. Whenever the IP layer receives an IP datagram to send, it determines which local interface the datagram is being sent on (routing), and queries that interface to obtain its MTU. IP compares the MTU with the datagram size and performs fragmentation, if necessary. Fragmentation can take place either at the original sending host or at an intermediate router.

When an IP datagram is fragmented, it is not reassembled until it reaches its final destination. (This handling of reassembly differs from some other networking protocols that require reassembly to take place at the next hop, not at the final destination.) The IP layer at the destination performs the reassembly. The goal is to make fragmentation and reassembly transparent to the transport layer (TCP and UDP), which it is, except for possible performance degradation. It is also possible for the fragment of a datagram to again be fragmented (possibly more than once). The information maintained in the IP header for fragmentation and reassembly provides enough information to do this.

Recalling the IP header (Figure 3.1, p. 34), the following fields are used in fragmentation. The *identification* field contains a unique value for each IP datagram that the sender transmits. This number is copied into each fragment of a particular datagram. (We now see the use for this field.) The *flags* field uses one bit as the "more fragments" bit. This bit is turned on for each fragment comprising a datagram except the final fragment. The *fragment offset* field contains the offset (in 8-byte units) of this fragment from the beginning of the original datagram. Also, when a datagram is fragmented the *total length* field of each fragment is changed to be the size of that fragment.

Finally, one of the bits in the flags field is called the "don't fragment" bit. If this is turned on, IP will not fragment the datagram. Instead the datagram is thrown away and an ICMP error ("fragmentation needed but don't fragment bit set," Figure 6.3) is sent to the originator. We'll see an example of this error in the next section.

When an IP datagram is fragmented, each fragment becomes its own packet, with its own IP header, and is routed independently of any other packets. This makes it possible for the fragments of a datagram to arrive at the final destination out of order, but there is enough information in the IP header to allow the receiver to reassemble the fragments correctly.

Although IP fragmentation looks transparent, there is one feature that makes it less than desirable: if one fragment is lost the entire datagram must be retransmitted. To understand why this happens, realize that IP itself has no timeout and retransmission—that is the responsibility of the higher layers. (TCP performs timeout and retransmission, UDP doesn't. Some UDP applications perform timeout and retransmission themselves.) When a fragment is lost that came from a TCP segment, TCP will time out and retransmit the entire TCP segment, which corresponds to an IP datagram. There is no way to resend only one fragment of a datagram. Indeed, if the fragmentation was done by an intermediate router, and not the originating system, there is no way for the originating system to know how the datagram was fragmented. For this reason alone, fragmentation is often avoided. [Kent and Mogul 1987] provide arguments for avoiding fragmentation.

Using UDP it is easy to generate IP fragmentation. (We'll see later that TCP tries to avoid fragmentation and that it is nearly impossible for an application to force TCP to send segments large enough to require fragmentation.) We can use our `sock` program and increase the size of the datagram until fragmentation occurs. On an Ethernet the maximum amount of data in a frame is 1500 bytes (Figure 2.1), which leaves 1472 bytes for our data, assuming 20 bytes for the IP header and 8 bytes for the UDP header. We'll run our `sock` program, with data sizes of 1471, 1472, 1473, and 1474 bytes. We expect the last two to cause fragmentation:

```
bsdi % sock -u -i -n1 -w1471 svr4 discard
bsdi % sock -u -i -n1 -w1472 svr4 discard
bsdi % sock -u -i -n1 -w1473 svr4 discard
bsdi % sock -u -i -n1 -w1474 svr4 discard
```

Figure 11.7 shows the corresponding `tcpdump` output.

```
1  0.0                   bsdi.1112 > svr4.discard: udp 1471

2  21.008303 (21.0083)   bsdi.1114 > svr4.discard: udp 1472

3  50.449704 (29.4414)   bsdi.1116 > svr4.discard: udp 1473 (frag 26304:1480@0+)
4  50.450040 ( 0.0003)   bsdi > svr4: (frag 26304:1@1480)

5  75.328650 (24.8786)   bsdi.1118 > svr4.discard: udp 1474 (frag 26313:1480@0+)
6  75.328982 ( 0.0003)   bsdi > svr4: (frag 26313:2@1480)
```

Figure 11.7 Watching fragmentation of UDP datagrams.

The first two UDP datagrams (lines 1 and 2) fit into Ethernet frames, and are not fragmented. But the length of the IP datagram corresponding to the write of 1473 bytes is 1501, which must be fragmented (lines 3 and 4). Similarly the datagram generated by the write of 1474 bytes is 1502, and is also fragmented (lines 5 and 6).

When the IP datagram is fragmented, `tcpdump` prints additional information. First, the output `frag 26304` (lines 3 and 4) and `frag 26313` (lines 5 and 6) specify the value of the identification field in the IP header.

The next number in the fragmentation information, the 1480 between the colon and the at sign in line 3, is the size, excluding the IP header. The first fragment of both datagrams contains 1480 bytes of data: 8 bytes for the UDP header and 1472 bytes of user data. (The 20-byte IP header makes the packet exactly 1500 bytes.) The second fragment of the first datagram (line 4) contains 1 byte of data—the remaining byte of user data. The second fragment of the second datagram (line 6) contains the remaining 2 bytes of user data.

Fragmentation requires that the data portion of the generated fragments (that is, everything excluding the IP header) be a multiple of 8 bytes for all fragments other than the final one. In this example, 1480 is a multiple of 8.

The number following the at sign is the offset of the data in the fragment, from the start of the datagram. The first fragment of both datagrams starts at 0 (lines 3 and 5) and the second fragment of both datagrams starts at byte offset 1480 (lines 4 and 6). The plus sign following this offset that is printed for the first fragment of both datagrams means there are more fragments comprising this datagram. This plus sign corresponds to the "more fragments" bit in the 3-bit flags in the IP header. The purpose of this bit is to let the receiver know when it has completed the reassembly of all the fragments for a datagram.

Finally, notice that lines 4 and 6 (fragments other than the first) omit the protocol (UDP) and the source and destination ports. The protocol could be printed, since it's in the IP header that's copied into the fragments. The port numbers, however, are in the UDP header, which only occurs in the first fragment.

Figure 11.8 shows what's happening with the third datagram that is sent (with 1473 bytes of user data). It reiterates that any transport layer header appears only in the first fragment.

Also note the terminology: an *IP datagram* is the unit of end-to-end transmission at the IP layer (before fragmentation and after reassembly), and a *packet* is the unit of data passed between the IP layer and the link layer. A packet can be a complete IP datagram or a fragment of an IP datagram.

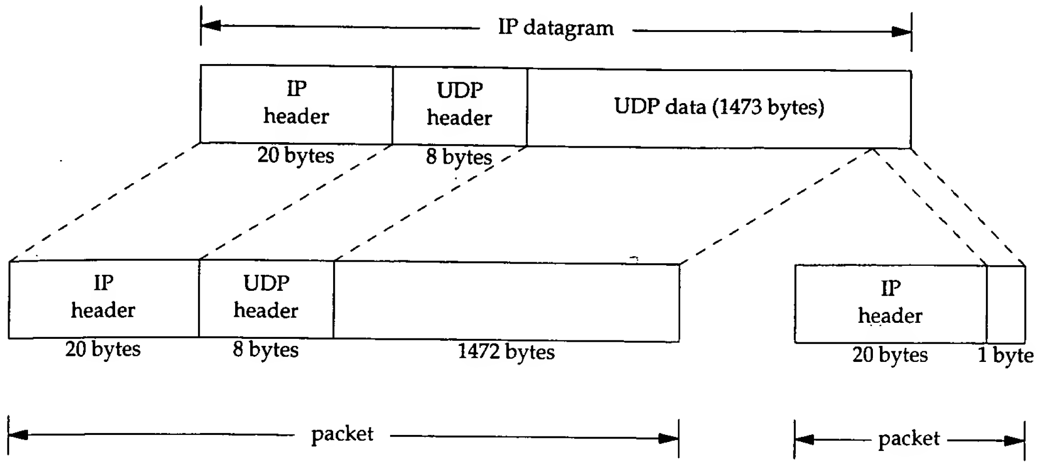

Figure 11.8 Example of UDP fragmentation.

11.6 ICMP Unreachable Error (Fragmentation Required)

Another variation of the ICMP unreachable error occurs when a router receives a datagram that requires fragmentation, but the don't fragment (DF) flag is turned on in the IP header. This error can be used by a program that needs to determine the smallest MTU in the path to a destination—called the *path MTU discovery* mechanism (Section 2.9).

Figure 11.9 shows the format of the ICMP unreachable error for this case. This differs from Figure 6.10 because bits 16–31 of the second 32-bit word can provide the MTU of the next hop, instead of being 0.

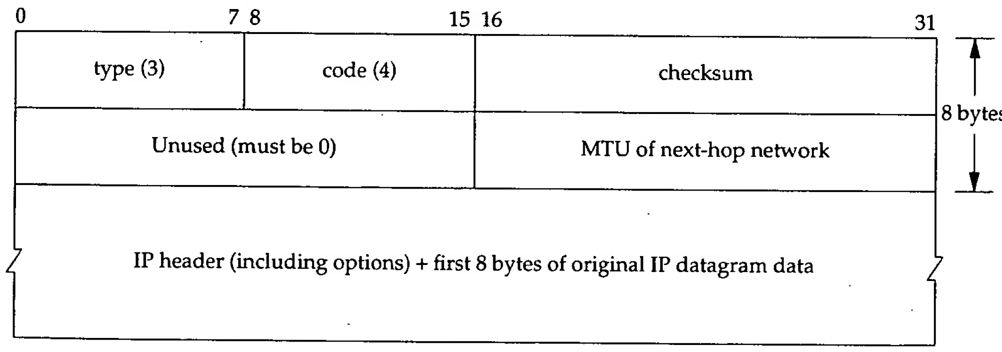

Figure 11.9 ICMP unreachable error when fragmentation required but don't fragment bit set.

If a router doesn't provide this newer format ICMP error, the next-hop MTU is set to 0.

The new Router Requirements RFC [Almquist 1993] states that a router must generate this newer form when originating this ICMP unreachable error.